Computationally Sound Mechanized Proofs of Correspondence Assertions

Bruno Blanchet
CNRS, Ecole Normale Supérieure
blanchet@di.ens.fr

July 2007
Our goal: implement an automatic, **computationally sound** prover for security protocols.

We have already implemented a prover for **secrecy properties**.

In this talk, we show how to extend it to **correspondence assertions**, that is, properties of the style:

\[
\text{If some event has been executed, then some other events have been executed.}
\]

Basic application: **authentication**.
The prover produces the proof is a sequence of games, as in Shoup’s or Bellare and Rogaway’s method:

- In the first game, the adversary plays against the real protocol.
- The prover transforms each game into the next by syntactic transformations or by applying security assumptions on cryptographic primitives.

Consecutive games are computationally indistinguishable.
- The desired security property can be proved directly on the last game.

Games are formalized in a process calculus.
Differences with secrecy

For the proof of correspondences:

- The language for games in the same as for secrecy, except for the addition of events.
- The game transformations and the proof strategy are the same as for secrecy. (The events are left unchanged.)
- One needs a new algorithm for checking correspondences on the last game.
Example: a nonce challenge

Simple example inspired by the corrected Woo-Lam public-key protocol (1997)

\[ B \rightarrow A : (N, B) \]
\[ A \rightarrow B : \{pk_A, B, N\}_{sk_A} \]

In our language:

```plaintext
c0(); new rk_A : keyseed;
let pk_A = pkgen(rk_A) in let sk_A = skgen(rk_A) in c1(pk_A);
  !i_A \leq n c2[i_A](xN : nonce, xB : host); event e_A(pk_A, xB, xN);
  new r : seed; c3[i_A](sign(concat(pk_A, xB, xN), sk_A, r))
  | !i_B \leq n c4[i_B](xpk_A : pkey); new N : nonce; c5[i_B](N, B);
  c6[i_B](s : signature); if verify(concat(xpk_A, B, N), xpk_A, s) then
  if xpk_A = pk_A then event e_B(xpk_A, B, N)
```

Bruno Blanchet
Computationally Sound Mechanized Proofs...
Arrays

All variables defined under replications are implicitly arrays. This allows us to store all values that occur during the executions of the game.
This replaces lists used by cryptographers and is key to cryptographic proofs.

c0(); new rk_A : keyseed;
let pk_A = pkgen(rk_A) in let sk_A = skgen(rk_A) in c1(pk_A);
\!i_A \leq n c2[i_A](xN[i_A] : nonce, xB[i_A] : host);
    event e_A(pk_A, xB[i_A], xN[i_A]); new r[i_A] : seed;
    c3[i_A]<sign(concat(pk_A, xB[i_A], xN[i_A]), sk_A, r[i_A])>
| \!i_B \leq n c4[i_B](xpk_A[i_B] : pkey); new N[i_B] : nonce; c5[i_B]<N[i_B], B>
    c6[i_B](s[i_B] : signature);
if verify(concat(xpk_A[i_B], B, N[i_B]), xpk_A[i_B], s[i_B]) then
if xpk_A[i_B] = pk_A then event e_B(xpk_A[i_B], B, N[i_B])
After game transformations using the unforgeability of signatures, the signature verification with $pk_A$ succeeds only for signatures generated with $sk_A$. After game transformations, we obtain the last game:

\[
\begin{align*}
c_0() & ; \textbf{new } rk_A : \text{keyseed} ; \textbf{let } pk_A = pkgen'(rk_A) \textbf{ in } c_1\langle pk_A \rangle ; \\
!i_A \leq n & \, c_2[i_A] (xN : \text{nonce}, xB : \text{host}) ; \\
\textbf{event } e_A (pk_A, xB, xN) & ; \textbf{let } m = \text{concat}(pk_A, xB, xN) \textbf{ in } \\
\textbf{new } r : \text{seed} & ; c_3[i_A] \langle \text{sign}'(m, skgen'(rk_A), r) \rangle \\
\mid & !i_B \leq n \, c_4[i_B] (xpk_A : pkey) ; \textbf{new } N : \text{nonce} ; c_5[i_B] \langle N, B \rangle ; \\
c_6[i_B] (s : \text{signature}) & ; \textbf{find } u \leq n \textbf{ such that } \\
\text{defined}(m[u], xB[u], xN[u]) & \land (xpk_A = pk_A) \land (B = xB[u]) \\
\land (N = xN[u]) & \land \text{verify}'(\text{concat}(xpk_A, B, N), xpk_A, s) \textbf{ then } \\
\textbf{event } e_B (xpk_A, B, N)
\end{align*}
\]
A non-injective correspondence is a formula of the form $\psi \Rightarrow \phi$ where

$$\phi ::= M \quad \text{term (without arrays)}$$

$$\text{event}(e(M_1, \ldots, M_m)) \quad \text{event}$$

$$\phi_1 \land \phi_2 \quad \text{conjunction}$$

$$\phi_1 \lor \phi_2 \quad \text{disjunction}$$

and $\psi$ is a formula that contains only events and conjunctions.

**Example**

$$\text{event}(e_B(x, y, z)) \Rightarrow \text{event}(e_A(x, y, z))$$

means that, if $e_B(x, y, z)$ is executed, then $e_A(x, y, z)$ has also been executed (except in cases of negligible probability).
Let ρ be an environment that maps variables to bitstrings. Let E be a sequence of events.

Definition

ρ, E ⊨ M if and only if M evaluates to true in environment ρ
ρ, E ⊨ event(e(M₁, ..., Mₘ)) if and only if
for all j ≤ m, Mⱼ evaluates to aⱼ in ρ and e(a₁, ..., aₘ) ∈ E

Definition

E ⊨ ψ ⇒ φ if and only if
for all ρ defined on var(ψ) such that ρ, E ⊨ ψ,
there exists an extension ρ' of ρ to var(φ) such that ρ', E ⊨ φ.

Definition

Q₀ satisfies ψ ⇒ φ with public variables V if and only if
for all evaluation contexts C accessing only variables of V in Q₀,
Pr[C[Q₀] executes E and E ⊨ ψ ⇒ φ] is negligible.
An injective correspondence also allows injective events
\( \text{inj-event}(e(M_1, \ldots, M_m)) \).

Each execution of the injective events in \( \psi \) corresponds to distinct injective events in \( \phi \).

Example

\[
\text{inj-event}(e_B(x, y, z)) \Rightarrow \text{inj-event}(e_A(x, y, z))
\]

means that each execution of \( e_B(x, y, z) \) corresponds to a distinct execution of \( e_A(x, y, z) \).
Intuition for the proof: non-injective correspondences (1)

Prove the correspondence \( \text{event}(e_B(x, y, z)) \Rightarrow \text{event}(e_A(x, y, z)) \)
in the game

\[ \ldots \! i_A \leq n \ldots \text{event } e_A(pk_A, xB, xN) ; \text{let } m = \ldots \text{ in } \ldots \]

| \! i_B \leq n \ldots \text{find } u \leq n \text{ such that } \text{defined}(m[u], xB[u], xN[u]) \land \\
\land (xpk_A = pk_A) \land (B = xB[u]) \land (N = xN[u]) \land \\
\land \text{verify}'(\text{concat}(xpk_A, B, N), xpk_A, s) \text{ then event } e_B(xpk_A, B, N) \]
Prove the correspondence \(\text{event}(e_B(x, y, z)) \Rightarrow \text{event}(e_A(x, y, z))\) in the game

\[
\ldots \downarrow i_A \leq n \ldots \text{event } e_A(pk_A, xB, xN); \text{let } m = \ldots \text{ in } \ldots
\]

\[
| \downarrow i_B \leq n \ldots \text{find } u \leq n \text{ such that defined}(m[u], xB[u], xN[u]) \wedge (xpk_A = pk_A) \wedge (B = xB[u]) \wedge (N = xN[u]) \wedge \text{verify}'(\text{concat}(xpk_A, B, N), xpk_A, s) \text{ then event } e_B(xpk_A, B, N)
\]

If \(\text{event}(e_B(x, y, z))\) has been executed, the program point \(\text{event } e_B(xpk_A, B, N)\) has been reached for some \(i_B = i'_B\), and \(e_B(x, y, z) = e_B(xpk_A[i'_B], B, N[i'_B])\).

So \(m[u[i'_B]], xB[u[i'_B]],\) and \(xN[u[i'_B]]\) are defined, \(xpk_A[i'_B] = pk_A, B = xB[u[i'_B]],\) and \(N[i'_B] = xN[u[i'_B]]\).

Since \(m[u[i'_B]]\) is defined, the definition of \(m[i_A]\) has been executed for \(i_A = u[i'_B]\), so \(\text{event } e_A(pk_A, xB[i_A], xN[i_A])\) has been executed.
If \( \text{event}(e_B(x, y, z)) \) has been executed, the program point \( \text{event} \ e_B(xpk_A, B, N) \) has been reached for some \( i_B = i'_B \), and \( e_B(x, y, z) = e_B(xpk_A[i'_B], B, N[i'_B]) \).

So \( m[u[i'_B]], x_B[u[i'_B]], \) and \( xN[u[i'_B]] \) are defined, \( xpk_A[i'_B] = pk_A, B = xB[u[i'_B]], \) and \( N[i'_B] = xN[u[i'_B]] \).

Since \( m[u[i'_B]] \) is defined, the definition of \( m[i_A] \) has been executed for \( i_A = u[i'_B] \), so \( \text{event} \ e_A(pk_A, xB[i_A], xN[i_A]) \) has been executed.

We have

- \( x = xpk_A[i'_B] = pk_A \)
- \( y = B = xB[u[i'_B]] = xB[i_A] \)
- \( z = N[i'_B] = xN[u[i'_B]] = xN[i_A] \)

so \( e_A(pk_A, xB[i_A], xN[i_A]) = e_A(x, y, z) \) has been executed.
Intuition for the proof: injective correspondences (1)

Prove the correspondence
\[ \text{inj-event}(e_B(x, y, z)) \Rightarrow \text{inj-event}(e_A(x, y, z)) \] in the game

\[ \ldots !^{i_A \leq n} \ldots \text{event } e_A(pk_A, xB, xN); \text{let } m = \ldots \text{ in } \ldots \]

\[ \mid !^{i_B \leq n} \ldots \text{find } u \leq n \text{ such that } \text{defined}(m[u], xB[u], xN[u]) \land \]
\[ (xpk_A = pk_A) \land (B = xB[u]) \land (N = xN[u]) \land \]
\[ \text{verify'}(\text{concat}(xpk_A, B, N), xpk_A, s) \text{ then } \text{event } e_B(xpk_A, B, N) \]
Prove the correspondence
\[
\text{inj-event}(e_B(i, x, y, z)) \Rightarrow \text{inj-event}(e_A(i', x, y, z)) \text{ in the game}
\]
\[
\ldots !^{i_A \leq n} \ldots \text{event } e_A(i_A, pk_A, x_B, x_N); \text{ let } m = \ldots \text{ in } \ldots
\]
\[
| !^{i_B \leq n} \ldots \text{find } u \leq n \text{ such that } \text{defined}(m[u], x_B[u], x_N[u]) \land (xpk_A = pk_A) \land (B = x_B[u]) \land (N = x_N[u]) \land \text{verify}'(\text{concat}(xpk_A, B, N), xpk_A, s) \text{ then event } e_B(i_B, xpk_A, B, N)
\]

In order to record in which session each event is executed, we add replication indices to events.
Prove the correspondence

\[ \text{inj-event}(e_B(i, x, y, z)) \Rightarrow \text{inj-event}(e_A(i', x, y, z)) \]

in the game

\[ \ldots !^{i_A \leq n} \ldots \text{event } e_A(i_A, pk_A, xB, xN); \text{let } m = \ldots \text{ in } \ldots \]

\| !^{i_B \leq n} \ldots \text{find } u \leq n \text{ suchthat } \text{defined}(m[u], xB[u], xN[u]) \land (xpk_A = pk_A) \land (B = xB[u]) \land (N = xN[u]) \land \]

\text{verify'}(\text{concat}(xpk_A, B, N), xpk_A, s) \text{ then event } e_B(i_B, xpk_A, B, N) \]

If \text{event}(e_B(i, x, y, z)) has been executed, the program point

\text{event } e_B(i_B, xpk_A, B, N) \text{ has been reached for some } i_B = i'_B, \text{ and } e_B(i, x, y, z) = e_B(i'_B, xpk_A[i'_B], B, N[i'_B]). \]

So \( m[u[i'_B]], x_B[u[i'_B]], \text{ and } xN[u[i'_B]] \) are defined, \( xpk_A[i'_B] = pk_A, \) \( B = xB[u[i'_B]], \) and \( N[i'_B] = xN[u[i'_B]] \).

Since \( m[u[i'_B]] \) is defined, the definition of \( m[i_A] \) has been executed for \( i_A = u[i'_B] \), so \text{event } e_A(i_A, pk_A, xB[i_A], xN[i_A]) \text{ has been executed.} \]
As before, $e_A(i_A, pk_A, xB[i_A], xN[i_A]) = e_A(i', x, y, z)$ for some $i'$.

In order to show injectivity, we show that

$e_B$ executed twice, for $i_B = i'_B$ and $i_B = i''_B$, with $i'_B \neq i''_B$

$\Rightarrow e_A$ executed twice, for $i_A = u[i'_B]$ and $i_A = u[i''_B]$, with $u[i'_B] \neq u[i''_B]$

By contraposition, we show $u[i'_B] = u[i''_B] \Rightarrow i'_B = i''_B$.

$u[i'_B] = u[i''_B] \Rightarrow xN[u[i'_B]] = xN[u[i''_B]]$

$\Rightarrow N[i'_B] = N[i''_B]$ since $xN[u[i'_B]] = N[i'_B]$ and $xN[u[i''_B]] = N[i''_B]$

$\Rightarrow i'_B = i''_B$ up to negligible probability
by eliminating collisions.
In order to prove $\psi \Rightarrow \phi$, two main steps:

1. **Collect the facts** that hold when the events in $\psi$ are executed.

2. Reason on these facts using an *equational prover* in order to show that the events in $\phi$ have been executed (and show injectivity when needed).

We shall now detail these points.
Collecting true facts

For each program point $P$, we collect a set of true facts at that point $\mathcal{F}_P$.

- We take into account assignments and tests above $P$.

**Example**

In `if M then P`, $M \in \mathcal{F}_P$.

- We take into account facts that hold at all definitions of variables.

**Example**

If `defined(x[\tilde{M}]) \in \mathcal{F}_P` and $M$ holds at all definitions of $x[\tilde{i}]$, then $M\{\tilde{M}/i\} \in \mathcal{F}_P$.

- We take into account that code is always executed up to the next output before switching to another thread.
Collecting true facts: example (1)

c0(); \textbf{new} \ rk_A : \text{keyseed}; \textbf{let} \ pk_A = pkgen'(rk_A) \textbf{ in } \overline{c1}(pk_A); \\
\!^{i_A \leq n}c2[i_A](xN : \text{nonce}, xB : \text{host}); \\
\textbf{event} \ e_A(pk_A, xB, xN); \textbf{let} \ m = \text{concat}(pk_A, xB, xN) \textbf{ in } \\
\textbf{new} \ r : \text{seed}; \overline{c3[i_A]}(\text{sign}'(m, skgen'(rk_A), r)) \\
| \!^{i_B \leq n}c4[i_B](xpk_A : \text{pkey}); \textbf{new} \ N : \text{nonce}; \overline{c5[i_B]}(N, B); \\
c6[i_B](s : \text{signature}); \textbf{find} \ u \leq n \textbf{ suchthat} \\
\text{defined}(m[u], xB[u], xN[u]) \land (xpk_A = pk_A) \land (B = xB[u]) \\
\land (N = xN[u]) \land \text{verify}'(\text{concat}(xpk_A, B, N), xpk_A, s) \textbf{ then } \\
\textbf{event} \ e_B(xpk_A, B, N) \\

At program point \( P = \textbf{event} \ e_B(xpk_A, B, N), \)

\( \mathcal{F}_P = \{ \text{defined}(m[u[i_B]]), \text{defined}(xB[u[i_B]]), \text{defined}(xN[u[i_B]]), \) \\
xpk_A[i_B] = pk_A, B = xB[u[i_B]], N[i_B] = xN[u[i_B]], \ldots \)
Collecting true facts: example (2)

c0(); new rk_A : keyseed; let pk_A = pkgen'(rk_A) in \overline{c1}(pk_A);

!^{i_A \leq n}c2[i_A](xN : nonce, xB : host);

\textbf{event} e_A(pk_A, xB, xN); let m = concat(pk_A, xB, xN) in

new r : seed; \overline{c3[i_A]}\langle \text{sign}'(m, skgen'(rk_A), r) \rangle

| ... |

At program point $P = \text{event} e_B(xpk_A, B, N)$,

$\mathcal{F}_P = \{ \text{defined}(m[u[i_B]]), \text{defined}(xB[u[i_B]]), \text{defined}(xN[u[i_B]]),$

$xpk_A[i_B] = pk_A, B = xB[u[i_B]], N[i_B] = xN[u[i_B]],$

\text{event}(e_A(pk_A, xB[u[i_B]], xN[u[i_B]])), ... \}$

because $\text{defined}(m[u[i_B]]) \in \mathcal{F}_P$. 
We use an algorithm inspired by the Knuth-Bendix completion algorithm to derive new equalities from known equalities.

The equational prover also eliminates collisions when they have negligible probability.

Details of this prover can be found in the paper: Blanchet, A Computationally Sound Mechanized Prover for Security Protocols, TDSC, to appear.

We say that \(\mathcal{F}\) yields a contradiction when the prover starting from \(\mathcal{F}\) derives false.
Proof of non-injective correspondences (1)

Let \( \psi \Rightarrow \phi = F_1 \land \ldots \land F_m \Rightarrow \phi \) be a non-injective correspondence, with fresh variables.

**Example**

\[
event(e_B(x, y, z)) \Rightarrow event(e_A(x, y, z)).
\]

If \( F_1, \ldots, F_m \) have been executed, then there exist \( P_1, \ldots, P_m \) such that, for all \( j \leq m \),
- \( F_j = event(e_j(M_{j1}, \ldots, M_{jm})) \),
- \( event\ e_j(M'_{j1}, \ldots, M'_{jm}); P_j \) occurs in \( Q_0 \), and
- \( event\ e_j(M'_{j1}, \ldots, M'_{jm}) \) has been executed with
  \( F_j = \theta'_j \cdot event(e_j(M'_{j1}, \ldots, M'_{jm})) \), where \( \theta'_j \) renames the replication indices at \( P_j \) to fresh replication indices.

**Example**

\[
event\ e_B(xpk_A, B, N) \text{ has been executed, with}
\]

\[
event(e_B(x, y, z)) = event(e_B(xpk_A[i'_B], B, N[i'_B])),\ \theta' = \{i'_B/i_B\}.
\]
Proof of non-injective correspondences (2)

Then the facts $\mathcal{F}_j = \theta'_j \mathcal{F}_{P_j} \cup \{\theta'_j M'_{j1} = M_{j1}, \ldots, \theta'_j M'_{jm_j} = M_{jm_j}\}$ hold.

Example

$\mathcal{F}_{P}\{i'_B/i_B\} \cup \{x = xpk_A[i'_B], y = B, z = N[i'_B]\}$ hold, where $\mathcal{F}_{P}$ has been described previously.

For each such $P_1, \ldots, P_m$, we show that

$$\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_m$$

implies $\theta \phi$ for some $\theta$ equal to the identity on $\text{var}(\psi)$, by the equational prover.

(For this proof, we prove atomic facts contained in $\theta \phi$, we choose $\theta$ by matching facts to prove with elements of $\mathcal{F}$, and we show that $\mathcal{F}$ implies $M$ by showing that $\mathcal{F} \cup \{\neg M\}$ yields a contradiction.)

Example

$x = xpk_A[i'_B], y = B, z = N[i'_B], xpk_A[i'_B] = pk_A, B = xB[u[i'_B]], N[i'_B] = xN[u[i'_B]], \text{event}(e_A(pk_A, xB[u[i'_B]], xN[u[i'_B]]))$ imply $\text{event}(e_A(x, y, z))$. 
We have also given:

- a proof of *mutual authentication* from correspondence assertions;
- a proof of *authenticated key exchange* from a combination of correspondence assertions and secrecy (also shown by our prover).

Difficulty: the secrecy is the secrecy of a single variable, the shared key is stored in two variables, $k_A$ in participant $A$, $k_B$ in participant $B$.

Intuitively, we show by correspondences that each key $k_B$ of $B$ with $A$ is an element of array $k_A$, so secrecy of $k_A$ is sufficient. Details in the paper.
Experimental results

We have tested our prover on the following protocols:

- Woo-Lam shared-key original and corrected versions
- Woo-Lam public-key original and corrected versions
- Needham-Schroeder public-key original and corrected versions
- Denning-Sacco public-key original and corrected versions
- Needham-Schroeder shared-key original and corrected versions, with and without key confirmation
- Yahalom, with and without key confirmation
- Otway-Rees

We try to prove authentication and authenticated key exchange, as appropriate for each protocol.

The prover obviously fails proving false properties.

It succeeds proving true ones, except in one case: the original version of the Needham-Schroeder shared-key protocol. (It does not see that $N_B[i] \neq N_B[i'] - 1$ except in cases of negligible probability.)
The prover succeeds proving most desired correspondences, but some points not directly related to correspondences could still be improved:

- **automatic proof strategy**: the prover sometimes needs indications from the user.
- **equational theories**, e.g. handle the equations of XOR.
- **handle more primitives**, e.g. Diffie-Hellman key agreements.

Tool available at

http://www.di.ens.fr/~blanchet/cryptoc.html